DOCUMENT RESUME

ED 144 612 IR 005 260

AUTHOR Larsson, Rolf; And Others
TITLE 3RIP: File Design for the Search System.
INSTITUTION Royal Inst. of Tech., Stockholm (Sweden). Library.
REPORT NO TRITA-LIB-4042
PUB DATE Mar 75
NOTE 41p.; For related documents, see IR 005 261 and IR 005 263.

EDRS PRICE MF-$0.83 HC-$2.06 Plus Postage.
DESCRIPTORS Bibliographic Citations; Computers; *Information Retrieval; Information Science; Information Storage; *Information Systems
IDENTIFIERS 3RIP Search System

ABSTRACT

The file design of the search system part of an interactive search and editing system, 3RIP, is described. A scatter-stored and compact inverted file is used to search a primary file of up to 4 million records containing on the order of 10-E9 characters of text and numeric data. Searchable attributes are keywords, words or phrases in text, names, and character strings, as well as values of numeric attributes. File updating costs can be estimated at o.o1-o.o2 Swedish crowns per typical record added to the primary file of the system. A typical record is assumed to contain 500 characters of textual and numeric data and some 50 directly searchable attributes generating entries in the inverted file. The amount of storage needed for the inverted file is approximately 30-50 per cent of what is needed for conventional inverted file systems. No more than two accesses are needed to reach an inverted file record with a known attribute, and some 95 per cent of the records are reached in one access. (Author)

****
* Documents acquired by ERIC include many informal unpublished *
* materials not available from other sources. ERIC makes every effort *
* to obtain the best copy available. Nevertheless, items of marginal *
* reproducibility are often encountered and this affects the quality *
* of the microfiche and hardcopy reproductions ERIC makes available *
* via the ERIC Document Reproduction Service (EDRS). EDRS is not *
* responsible for the quality of the original document. Reproductions *
* supplied by EDRS are the best that can be made from the original. *
****

ED144612

REPORT TRITA-LIB-4042 MARCH 1975

U S DEPARTMENT OF HEALTH, EDUCATION & WELFARE NATIONAL INSTITUTE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

3RIP: FILE DESIGN FOR THE SEARCH SYSTEM

Rolf Larsson, Christer Bryntesson, Bo Johansson and Mats Löfström

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Stephan Schwarz

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC) AND USERS OF THE ERIC SYSTEM."

IR005260

3RIP: FILE DESIGN FOR THE SEARCH SYSTEM

TABLE OF CONTENTS

Introduction 1
Characteristics of conventional inverted file systems 2
3RIP: File structure for the search system 8
3RIP: Logical operations on pairs of inverted file records 13
3RIP: Allocation of secondary storage and file partitioning 16
3RIP: File maintenance 19

Appendix A. 3RIP: Data structures for the inverted file 24
Appendix B. 3RIP: Hash and step algorithms 32
Appendix C. 3RIP: Simulation of the inverted file ... 35

References 37

ABSTRACT

The file design of the search system part of an interactive search and editing system, 3RIP, is described. A scatter-stored and compact inverted file is used to search a primary file up to 4 million records containing on the order of 10^9 characters of text and numeric data. Searchable attributes are keywords, words or phrases in text, names, and character strings as well as values of numeric attributes.

File updating costs can be estimated at 0.01-0.02 Swedish crowns per typical record added to the primary file of the system. A typical record is assumed to contain some 500 characters of textual and numeric data and some 50 directly searchable attributes generating entries in the inverted file.

The amount of storage needed for the inverted file is approximately 30-50 per cent of what is needed for conventional inverted file systems. No more than two accesses are needed to reach an inverted file record with a known attribute, and some 95 per cent of the records are reached in one access.

The computer used is a DEC System-10.

INTRODUCTION

With support from the Swedish Council for Scientific Information and Documentation (SINFDOK) the 3RIP system is being developed at the library of the Royal Institute of Technology in Stockholm. Two integrated parts, an interactive retrieval system for large files of text and numeric data, and an editing system are to constitute 3RIP. The editing system is not discussed in this report although its requirements to some extent have influenced the design of the search system which is being programmed on a DEC System-10 computer, a task to be completed by the end of 1975. The choice of computer has of course affected the structure of the system, but it is our belief that this report could form the basis for another implementation; as a single-processor system on a computer with good facilities for terminal handling and an architecture permitting efficient file and bit string processing, or as a multi-processor system where well defined and isolated functions of the system are assigned to different processors.

The report consists of three parts: a discussion of the shortcomings of interactive search systems based on the conventional file inversion technique when applied to large files; a presentation of the file organisation of the 3RIP search system which although based on file inversion is claimed to be well adapted to the processing of large files; and finally as appendices a more technical description of the file structures and the access techniques and some empirical results.

CHARACTERISTICS OF CONVENTIONAL INVERTED FILE SYSTEMS

In order to illustrate the file inversion technique a fairly typical, somewhat simplified, interactive bibliographic search system is examined. The system is assumed to contain N citations (bibliographic references to N documents). Moreover the number of recognized terms (different immediately searchable character strings, e.g. key words, author names, words in titles, etc.) is assumed to be T.

The system contains the following fundamental disc files:

BASE, the primary file consisting of the records BASE(i), i=1,2,...,N; where BASE(i) is the i-th citation (bibliographic reference to the i-th document).

BAD, an address vector with the elements BAD(i), i=1,2,...,N; such that BAD(i) indicates the location of the citation BASE(i).

TERM, consisting of the records TERM(i), i=1,2,...,T. TERM(i) contains a character string S(i), belonging to the set of recognized terms, and an address to the inverted file record IF(i) associated with S(i). TERM is alphabetically ordered so that if S(i) precedes S(j) in the collating sequence then $i < j$.
The file is moreover stored in D_0 blocks; there is a sequence of integers $0=t(0)<t(1)<\ldots<t(D_0)=T$, such that the records TERM(t(k-1)+1), TERM(t(k-1)+2),..., TERM(t(k)) constitute the k-th block of the file TERM.

DIR1, consisting of the records DIR1(i), i=1,2,...,D_0. DIR1(i) contains the character string S(t(i)) (i.e. the term string of the last record TERM(t(i)) of the i-th block of the file TERM), and the record number, i. DIR1 is stored in D_1 blocks, and there is a sequence of integers $0=s(0)<s(1)<\ldots<s(D_1)=D_0$, such that the records DIR1(s(k-1)+1), DIR1(s(k-1)+2),..., DIR1(s(k)) constitute the k-th block of DIR1.

DIR2, consisting of the records DIR2(i), i=1,2,...,D_1. DIR2(i) contains the character string S(t(s(i)))

(i.e. the character string of the last record DIR1(s(i)) of the i-th block of the file DIR1) and the record number, i.

IF, the inverted file proper, consisting of the records IF(i), i=1,2,...,T.
IF(i) represents the set of citations that contain or are associated with the character string S(i) of the record TERM(i). The representation is realized by a sorted vector of document or citation numbers, d(i,j), j=1,2,...,n(i), where n(i) is the number of elements in the set, and {BASE(d(i,j); j=1,2,...,n(i)} is the set of citations associated with S(i).

Using these files all citations associated with a given character string, S, can be found rapidly and with little difficulty. The following procedure might be used:

1. Search the directory file DIR2 sequentially for the first record DIR2(i) such that its character string S(t(s(i))) does not precede the string S in the collating sequence.

2. Get the i-th block of the next directory file, DIR1, and search for the first record DIR1(j) such that its character string S(t(j)) does not precede the string S in the collating sequence.

3. Get the j-th block of the file TERM and search for a string S(k)=S, where k = t(j-1)+1, t(j-1)+2, ... , t(j).

4. If no such string is found in the block the string S does not belong to the set of recognized terms.
If found get the document number vector
IF(k) = (d(k,1),d(k,2),...,d(k,n(k))) from the inverted file (TERM(k), which contains S(k), does also contain the address to IF(k)).

5. Retrieve the citation BASE(d(k,1)) from the location in BASE that is indicated by BAD(d(k,1)), 1 = 1,2,...,n(k).

If we assume that the top directory DIR2 can be kept in primary storage the number of accesses needed (a critical number in most interactive systems) for the retrieval of all citations associated with a given term would be 1+m(k)+2n(k); one access to get the appropriate directory block from DIR1, m(k) accesses to get the entire vector IF(k) (m(k) is the number of physical blocks in the file IF that contain IF(k)), and for each citation retrieved one access to the file BAD to get its location and one access to get itself.

However, it should be noted that in an interactive search system retrieval of citations only causes a small part of the total access load. Normally, a succesful search should not result in the retrieval of a large number of citations, and in the process of searching a fairly small number of citations is retrieved in order to test the validity of different ways of phrasing a query. Most of the access load is incurred by logical operations on pairs of records from the inverted file. Such operations require the steps 1-4 to be executed for each of the two records involved, and a new record is moreover generated, a result vector, structured as the inverted file records. The number of accesses needed for a logical operation on two inverted file records IF(k) and IF(l) will thus be 2+m(k)+m(l)+r, where m(k) and m(l) are the number of accesses needed to read the physical blocks containing IF(k) and IF(l), and r is the number of accesses needed to store the result vector.

The logical operations themselves are very simple (see e.g. Lefkovits (6)), and if the sizes of the two sorted citation number vectors constituting the inverted file records IF(k) and IF(l) are n(k) and n(l) respectively, the processor load caused by the operation can be estimated by (n(k)+n(l))c, where n(k)+n(l) is the number of comparisons needed to produce the result vector, and c is a system dependent constant (for a well designed system implemented on either of the large computers of Stockholms datamaskincentral, IBM 360-75 or DEC System-10, the value of c can be expected to lie between 5 and 15 microseconds).

When a new item (a new citation) is to be added to the system, those of its terms that are new must be added to the file TERM, the directory files must be modified accordingly, and each occurrence of a searchable term must moreover be added to the appropriate citation number vector of the inverted file. If the system is adapted to rapid retrieval, and no free space has been left in the blocks, or no suitable chaining mechanism is implemented, this implies a more or less complete regeneration of the files TERM, DIR1, DIR2, and IF.

If there is free space where needed, the updating process could be performed at much lower cost, but it is very difficult to predict where and what amount of space that will be needed. As a consequence of this the amount of free space must be relatively large and in spite of that the files have to be restructured fairly frequently, and the complexity of the file handling increases. If chaining is used the last statement is even more valid, and it is moreover to be expected that the number of accesses as well as the processor load when searching will be considerably higher than the corresponding values for a simpler system.

On-line updating might be allowed for a system with free space or with chaining, but in order to safeguard the integrity of the files and for economical reasons the files are normally updated in fairly infrequent batch runs, and the updating usually entails the creation of a new set of files. The cost of creating these new files (inverted file, term file, and directory files) grows approximately linearly with file size, however.

Based on the previous description of a conventional interactive bibliographic search system employing file inversion its characteristics may be classified summarily as follows (+ stands for good, = for acceptable, and - for poor characteristics).

Type of operation	File size=s (number of cit.)		
	$s \simeq 10^4$	$s \simeq 10^5$	$s \simeq 10^6$
Direct retrieval of a citation with known ordinal number	+	+	+
Logical operation on two inverted file records IF(i) and IF(j) consisting of n(i) and n(j) citation numbers, respectively;			
$n(i)+n(j) \simeq s/500$	+	+	+
$n(i)+n(j) \simeq s/50$	+	+	-
$n(i)+n(j) \simeq s/5$	+	-	-
File updating; addition of			
1 citation	-	-	-
s/100 citations	-	-	-
s/10 citations	+	-	-

Conventional interactive search systems based on file inversion deterioriate as the size of the inverted file increases. For small and medium-sized files this degradation can be disregarded, but not so with files of more than, say, half a million items. Each updating of the inverted file will for large files be very costly, and the logical operations on the larger records of the inverted file will consume a significant amount of processor time. Thus in searching the response time (with a given number of simultaneously active terminals) will increase with increasing file size, as will the cost of adding a given number of new records to the system.

It may be added that there are very few interactive bibliographic search systems that are used to handle files of one million citations or more, and that systems allowing searches in files of this size usually have partitioned the files into more manageable parts of perhaps a few hundred thousand citations each. This has obviously been done in order to keep the costs for file updating down and to keep the average response time to requests for logical operations at an acceptable level. One consequence of such a subdivision of the file is that the

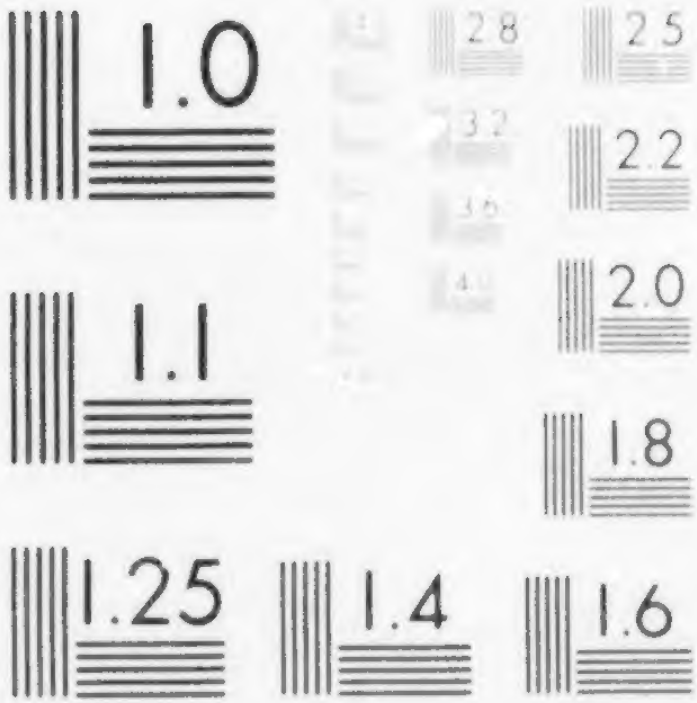
different parts of the file have to be searched one at a time, thus imposing a rather artificial restriction hampering any system user whose information needs cannot be satisfied within one such part, or who does not know which part to search in.

3RIP: FILE STRUCTURE FOR THE SEARCH SYSTEM

The primary file (of bibliographic references, other text items, product data, etc.) of the 3RIP search system is assumed to consist of N records, and the number of inverted file records in the system is assumed to be T. The values N and T should not exceed 4 million and 6 million respectively, but up to these values the performance of the system should be satisfactory; file maintenance costs will be low, and the cost of adding a given number of primary records to the system will be independent of N; costs associated with searching will also be low and compare favourably with those of conventional inverted file systems.

Each record of the inverted file is coupled with a different character string, its identifier, and the essential function of a record is to indicate the members of a set of primary file records with a common attribute (such as having the same author or being indexed with the same keyword). So far the 3RIP search system differs little from conventional inverted file systems. However, an inverted file record of 3RIP may also contain one or more values associated with each occurrence of its attribute in a primary file record.

Individual characteristics of each primary record indicated by an inverted file record may be given by attribute values. These may also be employed without introducing any redundancy, however. The primary file records associated with an attribute need not contain the attribute values; the attribute is indicated in the primary records, but the attribute values are stored in the appropriate inverted file record only. This latter use of attribute values is advantageous in applications where the values themselves and not the primary records are most important. In the text retrieval system STAIRS (4) we have an example of the first use of attribute values. There a record of the inverted file is associated with a word, and an occurrence of the word in the primary file is in the inverted file noted not only by the number of the primary file record but also by the location of the word in the record.



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

The second use of attribute values we can find in the data base management system ROBOT (8), which is designed to allow efficient processing, searching and updating of a data base where the structure of the primary file records remains fairly stable while the (numeric) attribute values frequently have to be processed (and modified).

After these preliminaries we can now turn to the fundamental files of the search system:

BASE, the primary file consisting of the variable length records BASE(i), i=1,2,...,N, stored in any order.

BAD, consisting of the records BAD(i), i=1,2,...,N, such that BAD(i) contains the address on secondary storage of the record BASE(i).

HEAD, containing the records HEAD(id(i)), where id(i) is the identifier coupled with the i-th attribute, and i=1,2,...,T.
HEAD(id(i)) contains all of the i-th inverted file record, or else one or more pointers into an overflow file where the inverted file record is stored.

BODY, the overflow file for inverted file record too large to be kept in the file HEAD.

All these files are stored in blocks of 512 machine words (of 36 bits), and in the case of the files BASE and BAD this causes no problems, but the handling of the inverted file, i.e. the files HEAD and BODY is not as trivial.

The file HEAD is divided into a number of head block sets, each consisting of 64 head blocks. Let us now consider the inverted file record associated with id(i), the character string identifying the i-th inverted file record.

A multiplicative hashcode generator with satisfactory distributive characteristics (see appendix B) maps the string id(i) onto a 44-bit hashcode, hash(id(i)). This bit string is then used to select a head block set, and to generate a sequence, hbn(id(i),j), j=0,1,...,15, of head block numbers within the head block set. The record HEAD(id(i)) is stored

in one block(hbn(id(i),j)), where j = 0, 1, ... , 14, or 15, and the value of j and the bit string hash(id(i)) are stored in block(hbn(id(i),0)), the home block of the i-th inverted file record.

Thus, when the inverted file record associated with the identifier id(i) is to be retrieved, the hashcode hash(id(i)) is generated, the proper head block set is identified, and block(hbn(id(i),0)) of the set is retrieved. If hash(id(i)) is not found in the block no inverted file record is associated with the identifier, else the value j, that was stored in the block together with the hashcode, is used as an index. If j=0, HEAD(id(i)) is stored in the same block as the hashcode, else hbn(id(i),j) is generated and block(hbn(id(i),j)) of the head block set is retrieved.

No more than two access operations are needed to get a record of HEAD, one access will do to verify the absence of a HEAD record, and one access will also be sufficient if the record is stored in its home block. According to results obtained when simulating the file structure about 95 per cent of the records of HEAD will be stored in their home blocks, when the load factor of the file is 0.95, i.e. when 5 per cent of the file space is unused (see appendix C).

The files above would be sufficient for a primitive retrieval system, but it would provide little help to a user who did not exactly know which identifiers to choose, or who were ignorant of the semantic relations between the identifiers of the system. In order to overcome these drawbacks and also to provide a simple means for string searching another file is introduced:

ALPHA, consisting of the records ALPHA(i), i=1,2,...,A.
Each record ALPHA(i) consists of a type indicator TYPE(i) followed by one character string, ALFA(i), and if indicated by TYPE(i) also by one or more of the strings MESSAGE(i), FATHER(i), BROTHER(i), and SON(i).
The file is alphanumerically sorted on the strings ALFA(i) and stored into B blocks of 512 machine words each. Each block is divided into 16 segments of 32

words in order to facilitate searching in and updating of the file; binary searching can be used to locate a segment, and new records can normally be added to the file without restructuring more than a segment or a block (free space is left when the file is created).

As the number of records in ALPHA can be expected to reach some 10 million, when the file BASE contains its maximal number of records, two directory files are needed:

BETA, consisting of the records BETA(i), i=1,2,...,B; where BETA(i) consists of the number i and the string ALFA of the last record of the i-th block of ALPHA. BETA is stored in C blocks of 512 words, and each block is divided into 16 segments of 32 words;

and

GAMMA, consisting of the records GAMMA(i), i=1,2,...,C; where GAMMA(i) is the last string of the i-th block of BETA.

When searching for a string s in ALPHA, s is compared with the records of GAMMA, a file small enough to be kept in primary storage, until the condition GAMMA(i-1)< s ≤ GAMMA(i) is met (inequalities refer to the order in the collating sequence). Then the i-th block of BETA is retrieved and comparisons are made until the condition beta(j-1) < s ≤ beta(j) is met (beta(k) is used to denote the string of the record BETA(k), k=1,2,...,B). Next the j-th block of ALPHA is read into primary storage and comparisons are made until ALFA(k-1) < s ≤ ALFA(k). If the string s is to be found in ALPHA then s=ALFA(k). Two accesses are thus needed to find the proper location in ALPHA for a given string.

The set {ALFA(i); i=1,2,...,A} contains all identifiers associated with the inverted file records, and the sets {FATHER(i); 1 ≤ i ≤ A}, {BROTHER(i); 1 ≤ i ≤ A}, and {SON(i); 1 ≤ i ≤ A} are subsets to it.

The strings FATHER, BROTHER, and SON are used to denote semantic relations, e.g. to establish a thesaurus where they might be associated with 'broader term', 'related term', and 'narrower term', respectively. The string MESSAGE in an ALPHA

record may be used to explain the usage of the ALFA-string of a record, but it may also be given another function (see appendix A).

Moreover the set ALFA(i); i=1,2,...,A contains a set of rotated strings. These are to be used for string searching in a subset of the set of identifiers, and they are produced from identifiers that are judged to contain significant substrings semantically related to the identifiers themselves (to allow for string searching in author names would obviously make little sense).

In order to illustrate the process the string SESQUIPEDALIAN is assumed to be an identifier to be rotated; then the strings ESQUIPEDALIAN*S, SQUIPEDALIAN*SE, QUIPEDALIAN*SES, UIPEDALIAN*SESQ, etc. would be inserted into the file ALPHA. A user of the system looking for records containing the string PEDAL might now request a string search to be performed. Then PEDAL would be matched against the directory files GAMMA and BETA, and the block containing the string PEDALIAN*SESQUI would be retrieved, and the original string SESQUIPEDALIAN would be sent to the user's terminal, where he would have to decide whether this occurrence of the string PEDAL could be of interest. If that were the case the inverted file record coupled with the string SESQUIPEDALIAN would be retrieved.

3RIP: LOGICAL OPERATIONS ON PAIRS OF INVERTED FILE RECORDS

The basic and also most common operations on the inverted file records of the 3RIP search system are assumed to be those associated with the logical operators AND, OR, and ANDNOT. Most more elaborate operations will involve one or more of these operators as components (adjacency conditions for example, only impose another restriction on a set defined by an and-operation).

The data structure of the inverted file of 3RIP (see appendix A) is logically equivalent to that of conventional inverted file systems, and a fairly simple interpretative subroutine may be used to generate the sorted vectors that are the physical equivalents of conventional inverted file records. Such a way of handling the records of the inverted file of 3RIP would enable the usage within the system of the wellknown and very simple algorithms for logical operations on records of conventional inverted files (see for example Lefkovits (6)).

The cost incurred by interpretation is for small inverted file records negligible, and for records corresponding to sorted vectors of some thousand elements it is still tolerable and moreover probably compensated for by the gains from not having to process any directory files in order to access the inverted file records. However, if large inverted file records frequently have to be interpreted in their entirety, this will cause a serious degradation of the performance of the system. Some other way of handling the large inverted file records is apparently needed unless the operations requiring their interpretation can be made fairly infrequent.

It can for the or-operation be argued that it rarely is very useful when applied to large inverted file records. The end result of any search should be a display of all or parts of a fairly small set of records of BASE, and moreover during a search only relatively few records of BASE can be examined. Thus the sets resulting from or-operations on large inverted file records must needs be transient and only be used as arguments in the process of defining the result set. One

consequence of this argument is that or-operations on large records rarely have to be executed. If either or both of a and b are large inverted file records, and a user request the operation (a OR b) to be performed, then the system might as well only respond with an estimate of the size of the set of BASE-records corresponding to the result of the operation. If the user in some later interaction requests the result of this or-operation to be intersected with another inverted file record c, the system would respond by doing the intersections (a AND c) and (b AND c), and only if the resulting sets were small enough would the or-operation ((a AND c) OR (b AND c)) be done, otherwise only an estimate of the set size would be made and passed to the user. It should be noted that the user need not be made aware of the deferral of the or-operations, as this would be internal to the system and manifest only in the inaccuracy of the estimate of some of the sets of records of BASE that were contemplated during a search.

Thus the problem of efficiently handling large inverted file records in intersecting operations remains. If the records a and b are to be intersected and only b is large, the record a can be interpreted and the elements of the resulting sorted vector may be used as arguments of a simple lookup procedure (the inverse of the interpreting procedure) locating their counterparts in b. If both records are large, three different methods are being considered: if the need of the particular operation (a AND b) can be anticipated it might be made part of the file updating process, and an inverted file record corresponding the result of the operation would be created; a sampling technique might be employed to get an estimate of the size of the set resulting from the operation and to get some, if any, records of BASE belonging to the set, thus enabling the user to decide whether the execution of the operation would be worth the cost that he would be charged for; and finally algorithms designed to operate directly on the compact storage structures that are used for the inverted file records (see appendix A). Such algorithms have been developed but they need to be modified and improved on in order to be used in the 3RIP search system. The algorithms are based on the fact that large inverted file records can be stored

efficiently as bit vectors where each bit corresponds to a record of BASE (see for example Byrom and Hardgrave (1), who in their paper also present some algorithms applicable to a subset of the structures that are considered for the inverted file of the 3RIP search system).

3RIP: ALLOCATION OF SECONDARY STORAGE AND FILE PARTITIONING

The files of the 3RIP search system should, for the system to perform satisfactorily, be kept on secondary storage devices with relatively short access time, e.g. normal moving head disc units. In order to keep the cost for secondary storage of the larger files of the 3RIP search system at an acceptable level two standard methods may be applied: data compression or generation of different sets of data bases made available during different time slots.

In a separate report (3) on work done within the 3RIP project a system for text compression is presented. There it is shown that the storage requirements for data from a wide range of bibliographic data bases can be reduced to an average of 2.9 - 3.2 bits per character of textual data (titles, keywords, names, etc.). The cost of compacting the data can be estimated at 75 Swedish crowns per million characters processed. In order to expand the coded data to their original form a decoding dictionary of about 4 K machine words (36 bits) is needed. This dictionary should be resident in primary storage in order to enable rapid decoding; on the DEC 10 computer to be used the decoding process is estimated to require less than 10 microseconds of processor time per generated character. If this method is used to compress the file BASE, its storage requirements can be expected to be halved (the space needed for the textual parts is reduced to about 40 per cent, but control information is still needed to make the individual fields of the records of BASE easily accessible). The cost of decoding retrieved records of BASE will be almost negligible in normal usage of the 3RIP search system.

The amount of secondary storage needed for the inverted file of 3RIP will vary with the amount of information that is associated with each occurrence of an attribute coupled with a record of BASE. If no more than a record number is needed, then, according to an earlier 3RIP report (5), 11 - 12 bits per occurrence will suffice. If attribute values or locations within the individual records of BASE are to be noted in some

inverted file records the amount of storage space needed will increase somewhat, however.

If BASE is assumed to contain 10^9 characters in 10^6 records, the total number of attribute occurrences is assumed to be 10^8, and each such occurrence requires an average of 20 bits of storage space, then the files BASE, HEAD, and BODY of the 3RIP search system would require a total of about $6 \cdot 10^9$ bits, and the entire system would require about 150 million words, or about a third of what would be needed for a retrieval system such as STAIRS (4).

In order to use the search system all of the files HEAD, ALPHA, BETA, and GAMMA must be available on secondary storage. The files BASE, BAD, and BODY on the other hand can be partitioned, however, if the amount of available disc storage is insufficient for the entire data base of the 3RIP search system. In order to facilitate this partitioning the following constraints should be imposed on the file updating process:

1. The total file BASE should be divided into 2^7 <u>physical subfiles</u> $B_1, B_2, \ldots, B_{128}$; such that the records $BASE(2^{15(i-1)}+1)$, $BASE(2^{15(i-1)}+2)$, ... , $BASE(N_i)$, where $N_i \leq 2^{15i}$, constitute the subfile B_i.

2. The elements of a physical subfile of BASE should all belong to one <u>logical subfile</u> which for example might be oriented to one discipline.

3. When updating a logical subfile its new BASE-records should be added to its last physical subfile, and the new records of BASE, BAD, and BODY should all be assigned to the same physical storage units.

The system could then be run with the files HEAD, ALPHA, BETA, GAMMA, and any combine of physical storage units assigned to the files BASE, BAD, and BODY.

Another important consequence of a file partitioning according to the rules above will be that a user easily can limit any parts of a search to one or more logical subfiles. Within the

system such a limitation is performed by a simple and-operation between each HEAD-record accessed by the user and a bit-mask of 128 bits defining the physical subfiles that are assigned to the logical subfiles he is interested in.

3RIP: FILE MAINTENANCE

The files of the 3RIP search system are normally to be updated in batch mode, but facilities for on-line updating are to be provided within the 3RIP editing system. The development of the on-line updating routine is not expected to cause any serious problems, as one of the primary objects of the file design has been to keep the effects of modifications in one file from propagating through the system. In this report only batch mode updating will be discussed, however.

Let us regard the case of adding a file of new records, BASEADD, to the primary file BASE, which is assumed to contain N records. In order to be able to ignore elements judged to be of little significance and also in order to reduce the volume of the auxiliary files that are needed, a new permanent file is introduced:

COMMON, consisting of the records COMMON(i), i=1,2,...,n; where $5000 < n < 20000$. The constituents of COMMON(i) are a type indicator and a character string belonging to either of two sets: the set of identifiers associated with the largest inverted file records, or a set of stop words, i.e. words or phrases deemed to have little semantic content in themselves. Membership in the sets is given by the type indicator.

The file updating process can now be summarized as follows:

1. Read the entire file COMMON into primary storage scattering it in order to facilitate rapid searching in the file.

2. For each record of BASEADD perform the steps a and b:

   a. N:=N+1; generate the record BASE(N) from the current record of BASEADD and store it into the file BASE; BAD(N):=the address of BASE(N).

   b. Match against the file COMMON those fields of the current record that are to be searchable; ignore strings found to match members of the set of stop words; write a record onto an auxiliary sequential file, CNEW, for each string matching the set of com-

mon identifiers; and write another record onto the auxiliary file ANEW for each unmatched string whose presence in the current record is to be a searchable attribute.
A record of CNEW contains the hashcode (that the matched identifier is mapped onto in the inverted file), the record number N, and in some cases, depending on the type of the identifier, its location in the record BASE(N) (e.g. sentence number and word number within sentence) or some other numeric value associated with the occurrence of the identifier.
A record of ANEW is similar to one of CNEW, but instead of hashcode it contains the unmatched identifier (e.g. in a text field a word not belonging to the set of common words or the set of stop words) and a boolean variable, set to true if the file ALPHA should contain the rotated forms of the identifier (i.e. if its substrings are to be accessible), and finally it contains the record number N and possibly also the location of the identifier in BASE(N) or a numeric value associated with it.

3. Sort CNEW with hashcode and record number as primary and secondary sort keys (and possibly with location within record of BASE or associated numeric value as a tertiary sort key).

4. Sort ANEW with identifier and record number as primary and secondary sort keys, respectively (and possibly with location or value as tertiary key).

5. Reading the files ANEW and ALPHA in parallell execute the steps a-c for each different identifier of ANEW:

   a. Create a new record consisting of the hashcode associated with the identifier, a boolean variable M set to false, and for each record of ANEW containing the current identifier an element giving its record number in BASE (and possibly also location within record of BASE or numeric value).

   b. If the identifier is not found in ALPHA, insert it

at its proper location and set to true the boolean variable M of the record created in the previous step. Moreover, if ALPHA is to contain the rotated forms of the string, generate those and insert them at their proper locations in ALPHA.

c. Write the record created in a. onto the auxiliary file BNEW.

6. Create the auxiliary file DNEW from the sorted file CNEW. The record format of DNEW is to be that of BNEW, with the boolean variable set to false.

7. Merge the files BNEW and DNEW creating the auxiliary file NEW, where a record consists of a boolean variable M, a hashcode, and a posting set, a vector of BASE record numbers (possibly appended with locations within the records of BASE or with attribute values) from all the BNEW or DNEW records containing the hashcode. M in a record of NEW is assigned the value of its counterparts in BNEW or DNEW.

8. Sort NEW on the hashcode part denoting head block set (see appendix A) and within head block set descendingly on the size of the posting set.

9. Use NEW to update the files HEAD and BODY. Perform the steps a and b for each head block set represented in NEW.

   a. Read the head block set into primary storage.

   b. Perform the steps c-f for each record of NEW associated with the head block set.

   c. Generate the sequence b(j), j=0,1,...,15 of block numbers associated with the hashcode of the record (see appendix B).

   d. If M of the record is true, then store the hashcode in block(b(0)) of the block set.

   e. Create the proper compact representation of the posting set of the record if M is true, else create the representation of its union with the posting set associated with its hashcode in the inverted file.

   f. Store the resulting representation in the home block (i.e. block(b(0))) if there is enough free space,

else store it in the block(b(j)), j=0,1,...,15, with most free space.

The data structure for the posting set representation generated in step 9e is chosen among those defined in appendix A, and the structure requiring least space is used. If the size of the representation exceeds a predetermined value, the representation is stored in one or more blocks of the file BODY, a pointer structure indicating the location and size of the different portions of the posting set is created, and in the algorithm the representation of the posting set is exchanged for this pointer structure.

When a large posting set is being updated it should be noted, however, that usually no more than one of its earlier body blocks need be involved; only the last body block referred to by the pointer structure of the old posting set is modified, new postings are appended and if more space is required, another body block is assigned to the posting set and a new pointer is added to the pointer structure.

When the load factor of a head block set (the quotient between the amount of used space and the total amount of space in the head block set) approaches 1, then there will in some cases be no free space available for the storage of a new hashcode in its home block or for the representation of a posting set in any of its head blocks numbered b(j), j=0,1,...,15. Then in order to create free space for the representaion one or more representations requiring less space than the current one are removed from the block(b(j)) with the largest amount of free space; and in order to create space for the hashcode one small representation of a posting set is removed. Each of the removed representations are later stored in their home block or in that of their 15 other head blocks with most free space when they might cause the removal of smaller representations which then have to be stored into the block set. However, according to results obtained when simulating the inverted file (see appendix C) removals to create space will be fairly rare even if the load factor is close to 1, and most representations will moreover be stored in their home blocks.

As the size of the inverted file increases the file HEAD must eventually be restructured. When this is done the number of head block sets is doubled as each record of a head block set is stored into either of two new head block sets. It should be noted, however, that no other files have to be involved and that the same 44-bit hashcode as before will be used.

The cost of the file updating process will, for a given size of BASEADD, remain fairly stable while the size of the inverted file increases. It is probably advantageous to execute some of the steps above (1, 2b, 3, and 4) on a large batch-oriented computer. If that is done, the total file updating cost per added BASE record (of on the average 500 characters and generating 50 postings in the inverted file) is estimated to 0.01-0.02 Swedish crowns. The estimate is based on the computer charges as Stockholms datamaskincentral.

A P P E N D I X A

3RIP: DATA STRUCTURES FOR THE INVERTED FILE

HEAD

File size: 2^h head block sets; $2 \leq h \leq 10$;
Head block set size: 64 head blocks;
Head block size: 512 words (36 bits).

Each record of HEAD is associated with an identifier and a posting set. An identifier is a character string used to denote a searchable attribute, and no two identifiers in 3RIP may be alike. A posting set indicates all occurrences of a given attribute in or associated with the records of the file BASE. A posting consists of a record number in 22 bits (indicating a record of BASE) and for some posting sets another bitstring giving the location of the attribute within its BASE record or else containing a value of the attribute.

In order to find the proper location for a record of HEAD the associated identifier is mapped onto a 44-bit hashcode, H (see appendix B). The hashcode consists of five concatenated bit strings: m in h bits, n in 6 bits, p in 4 bits, q in 24 bits, and r in 16-h bits.

The location of the record is then to be in the m-th head block set, and within that set in the n-th block and possibly also in one of a sequence of 15 other head blocks of the set. The sequence is decided by a step algorithm based on the bitstrings m and r (see appendix B).

In the n-th block q is stored at the location indicated by the p-th element of an index vector. q is stored in the leftmost 24 bits of a 36-bit word, and the index vector is stored in the last four words of the n-th block.

This may be illustrated as follows:

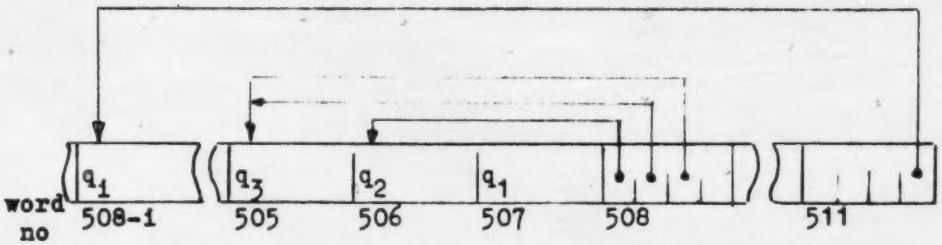


Fig. The right part of a head block where i hashcodes have been stored. Two hashcodes with p=0, one with p=1, and none with p=2 is indicated. The first free word to the left of the hashcodes is word 507-i of the block.

The rightmost 12 bits of each word containing a q value are used for an 8-bit internal step counter and a 4-bit external step counter indicating the location of head, the remainder of the record. In head all or part of a complete representation of the posting set associated with the current inverted file record is stored. In the latter case head contains one or more pointers to a complete representation of the posting set. This representation is denoted body and it is stored in one or more body parts in the file BODY.

Set b(0)=n, and let b(1), b(2), ... , b(15) be the sequence of block numbers that are generated by the step algorithm. head is then stored in one of the blocks numbered b(j), j = 0, 1, ..., 15; and the value of j is stored in the external step counter.

To facilitate accessing of the heads associated with different inverted file records a head index vector is set up in each head block preceding the heads of the block. The internal step counter points to the appropriate vector element which points to the head. The vector elements are 18 bits each; 12 bits addressing the head relative to the beginning of the block, and a 6-bit head type indicator.

The first bit of a head type indicator, the home flag is set if head is not stored in its home block. The second bit, the body flag, is set if head contains one or more pointers to the body of the inverted file record. The third bit, the message flag, is set if head contains a message field.

The last three bits, the span of a head, indicate the extent of the spanner, a bit vector or equivalent structure describing the high-order part of all postings of the inverted file record.

The size of the high-order parts described by the spanner may be 0, 3, 7, 11, 15, 19, or 22 bits which is indicated by a span value of 1, 2, 3, 4, 5, 6, or 7 respectively. span=0 indicates the absence of a posting set.

If the body flag is not set, i.e. if the entire posting set is represented in head, a span value from 2 to 6 indicates that head contains one chain for each of the different values of the high-order parts of the postings of the inverted file record, and a span value equal to 1 indicates one chain.

The low-order parts of postings with the same high-order parts constitute a chain.

h e a d

A head is stored in a sequence (of no more than 2000) contiguous 4-bit elements of a head block. From left to right its constituents are:

home block number,	if the home flag = 1;
r of the hashcode,	if $h < 16$;
message field,	if the message flag = 1;
spanner,	if span > 1;
pointer part,	if the body flag = 1;
chain part,	if the body flag = 0 and $1 \leq$ span < 7.

If a head is not stored in its home block then the number within the block set of its home block is stored in an 8-bit home block number.

The bitstring r is stored in 4 or 8 bits depending on the size of h.

It should be noted that information in the message field of a head might imply a modified or altogether different interpretation of data following the message in a head.

head: m e s s a g e f i e l d

The message field contains one or more messages to be dispatched when the head enclosing the field is to be processed. Messages are used to activate processes within the system and/or to assign values to accessible variables of the system.

Examples of messages might be:

- the identifier associated with the record; such a message might be used in order to separate identifiers known to have the same hashcode.
- a password or a set of passwords to be sent to a sentry process guarding against unauthorised usage of the posting set.
- a notice to be sent to the terminal requesting the head to be processed.
- data modifying the declaration of the spanner and/or the pointer or chain part of the representation of the current posting set.

head: s p a n n e r

If not modified by a message the spanner corresponds to a bit vector of 2^s bits; s = 0, 3, 7, 11, 15, 19, or 22 corresponding to span = 1, 2, 3, 4, 5, 6, and 7, respectively.

If s = 0 no spanner structure is explicitly stored in the head (a spanner of one bit = 1 is implied, however), else it may be given an immediate or an enumerative representation, or if s>3 a bit stream set representation or a segment representaion may be employed. The representation type is decided by the declaration of the file HEAD, but it may be altered by a message.

head: spanner: i m m e d i a t e r e p r e s e n t a t i o n

The bit vector corresponding to the spanner is stored in head.

head: spanner: e n u m e r a t i v e representation

The ordinal numbers of the bits set to one (or of those set to zero, if specified by a message) are stored in head.

head: spanner: b i t s t r e a m s e t representation

The spanner is represented by a sequence of bit vectors (directories) $B_0D_1 \ldots D_n$ generated recursively from the bit string

$$B_n = b_{n0}b_{n1}\ldots b_{ns_n}$$

coinciding with the immediate representation of the spanner. In order to generate the directories the following definitions are introduced:

b_{ij} denotes a binary digit: $i = 1,2,\ldots,9$; $j = 0,1,\ldots,s_i$; $s_i = 2^{3+2i}-1$ for $i = 1,2,\ldots,8$, and $s_9 = 2^{22}-1$.

$$B_{ij} = \begin{cases} b_{i,4j}b_{i,4j+1}b_{i,4j+2}b_{i,4j+3} & \text{for } i = 1,2,\ldots,8 \\ b_{i,4j}b_{i,4j+1} \cdot \cdot \cdot b_{i,4j+7} & \text{for } i = 9 \end{cases}$$

$$B_i = B_{i0}B_{i1} \cdot \cdot \cdot B_{is_{i-1}}$$

$$\emptyset_1 = \emptyset_2 = \ldots = \emptyset_8 = 0000 \quad \text{(binary digits)}$$

$$\emptyset_9 = 00000000 \qquad \text{- " -}$$

$$D_{ij} = \begin{cases} \text{the empty string if } B_{ij} = \emptyset_i \\ B_{ij} \text{ else} \end{cases}$$

$$D_i = D_{i0}D_{i1} \cdot \cdot \cdot D_{is_{i-1}}$$

$$b_{i-1,j} = \begin{cases} 0 \text{ if } B_{ij} = \emptyset_i \\ 1 \text{ else} \end{cases}$$

Using this notation the bitstring $B_{n-1}D_n$ is an unambiguous representation of B_n (the immediate representation of the spanner) with the 4-bit (or 8-bit) segments of B_n indicated by corresponding bits in B_{n-1} set to zero.

In the same manner $B_{n-2}D_{n-1}D_n$ represents $B_{n-1}D_n$ unambiguously etc., and finally the representation $B_0D_1 \ldots D_n$ is reached.

head: spanner: s e g m e n t representation

The immediate representation of the spanner is regarded as a string of n 4-bit elements,

$$H = h_0 h_1 \ldots h_{n-1}$$

A <u>proper null string</u> of H is now defined as substring of H,

$h_i h_{i+1} \ldots h_{i+j-1}$, such that $i > 0$, $j \geq 3$, $i+j < n$,

$$h_i = h_{i+1} = \ldots = h_{i+j-1} = 0000$$

$$h_{i-1} \neq 0000, \; h_{i+j} \neq 0000$$

The m-th proper null string encountered while scanning H from left to right is denoted N_m.

Assuming the number of proper null strings of H to be s, N_0 and N_{s+1} are defined as the (possibly empty) strings of initial and final 4-bit elements = 0000 of H.

Use S_m to denote the substring of 4-bit elements between N_m and N_{m+1}. $m = 0,1,\ldots,s$.

Use k_m to denote the number of 4-bit elements constituting N_m.

Use l_m to denote the number of 4-bit elements constituting S_m.

Let K_m and L_m be minimal strings of 1, 2, 3, 4, or 5 4-bit elements representing the values k_m and l_m, respectively.

Then the segment representation of the spanner is given by the sequence

$$K_0 L_0 S_0 K_1 L_1 S_1 \ldots K_{s-1} L_{s-1} S_{s-1} K_s L_s S_s.$$

head: p o i n t e r p a r t

If all of the posting set of the inverted file record is not stored in its head, it is distributed over one or more blocks of the file BODY with pointers indicating the different parts of the posting set, their sizes and their location in BODY.

Each pointer consists of,

<u>last record set</u>,	in 12 bits;
<u>approximated posting count</u>,	in 6 bits; and
<u>body block number</u>,	in 18 bits.

The file BASE is assumed to be divided into no more than 4096 record sets; the first record set is assumed to contain the first 1024 records, the second the next 1024, etc. The last record set in a pointer denotes the ordinal number of the last record set that is stored in the body part referred to.

Let the number of postings represented by the k-th body part be n(k). This value is approximated by the value closest to n(k) of the values $a^1, a^2, \ldots, a^{64}$, where $a = 2^{14/64} \approx 1.164$, and the exponent is given by the approximated posting count of the k-th pointer.

head: c h a i n p a r t

The chain part consists of a sequence of chains in the same order as their indicators in the spanner. Thus the i-th bit of the bitstring represented by the spanner is associated with the i-th chain (if span = 1, a spanner of one bit = 1 is assumed).

Each chain consists of one or more links, and each link consists of

chain end flag, in 1 bit; and
link value, in 4(7-span)-1 bits.

The chain end flag is set to zero for all links but the last of each chain.

BODY

File size: no more than 2^{18} body blocks
Body block size: 512 words (36 bits)

Each body block contains one or more body parts indicated by pointers in heads of the file HEAD. In the body blocks the body parts are stored from left to right, and each body part is assigned a sequence of contiguous 4-bit elements.

The constituents of a body part are:

body part size, in 8, 12, or 16 bits;
hashcode, in 44 bits;
message flag, in 1 bit;
span, in 3 bits; and
body member, in the remainder of the body part.

The body part size represents the number of 4-bit elements constituting the body part, and as different body parts of a body block form a contiguous sequence the size values are used to facilitate accessing of the different body parts.

The message flag and the span indicate the structure of the body member, which consists of,

message field, if the message flag = 1;
spanner, if span > 1; and
chain part, if $1 \leq$ span < 7.

The constituents of the body member are to be interpreted as their counterparts in the heads of the file HEAD.

A P P E N D I X B

3RIP: HASH AND STEP ALGORITHMS

Every identifier to be a member of HEAD is transformed to a hashcode. From the hashcoding algorithm and a step algorithm it is possible to compute the 16 head blocks in HEAD associated with the identifier. These algorithms are presented together with some results obtained from tests with identifiers from the MEDLINE system. We use the following notation:

m: number of machine words used to store the identifier concatenated with a string of pad characters.

f: machine word size in bits.

w_i: the bitstring constituting the i:th machine word. i=1,...,m

pr: product stored in two machine words.

pr_{low}: the f low order bits of pr.

pr_{high}: the f high order bits of pr.

Let **F(a,k)** be a function which gives as a result a bitstring constituting the centermost k bits of the bitstring a.

Then the hashcoding algorithm, which produces a hashcode B consisting of b bits can be described.

pr:= w_1 · w_m ;
i:=1;
while i ≥ m-2 do
begin
  cp:=F(pr,f);
  pr:= cp · w_{i+1};
  i:= i + 1
end;
pr:= pr_{low} · pr_{high};
B:= F(pr,b);

The hashcoding algorithm should be such that it, for a given set of identifiers, produces a set of hashcodes, where the number of collisions, i.e. the number of different identifiers minus the number of hashcodes produced, is minimized. Theoretically, assuming that the hashcodes are independent random numbers drawn from the integers $0,1,\ldots,2^b - 1$, the number of collisions CO is computed by Murray (7). For a given b and N different identifiers we get the expected number of collisions as

$$CO = N - 2^b(1 - e^{-N/2^b})$$

The hashcoding algorithm presented above has been used on the identifiers of MEDLINE with N = 536778. The number of collisions for different values of b has been compared with the theoretically computed value and the result is shown below. For b > 35 no collisions occured in the MEDLINE case.

b	25	26	27	28	29	30	31	32	33	34	35
$CO_{MEDLINE}$	4255	2083	1027	515	240	122	74	35	15	7	3
$CO_{theoretical}$	4270	2141	1071	536	268	134	67	33	16	8	3

We use this result to support the assumption that hashcodes produced from a set of N different textual character strings can be regarded as N independent random numbers.

It is then possible to estimate the probability of getting at least one collision with a function of b and N:

$$P(col) = P(\text{at least one collision}) = 1 - e^{-N^2/2^{b+1}}$$

In the 3RIP-system b is chosen to 44 bits. Then for different N we get:

N millions	0.5	0.75	1	2	4
P(col)	0.007	0.016	0.028	0.108	0.265

This is a sufficiently low probability of collisions, even when the number of identifiers grows.

In appendix A it is shown how parts of the hashcode are used to locate the head block numbered b(0), the home block of the identifier. The sequence of block numbers b(1),b(2), ...,b(15) also associated with the identifier is computed by a step algorithm, similar to the one presented in (2). The following notation is used:

M: number of head blocks in a head block set.

P: the least prime number, which is greater than or equal to M.

From the hashcode a number of bits are used to compute the value of a parameter S where $1 \leq S < M$. Then the step algorithm can be described as follows:

```
for i:= 0 step 1 until 15 do
begin
   while M < b(i) + S ≤ P do b(i):= b(i) + S;
   if b(i) + S ≤ M then b(i+1):= b(i) + S else
   b(i):= b(i) + S - P;
end;
```

APPENDIX C

SIRIP: SIMULATION OF THE INVERTED FILE

A simulation model giving a close description of the updating of one head block set has been developed. The model has been programmed in SIMULA and is run interactively on a DEC-10 computer. Results from experiments with the model have been used in the design of the inverted file structure and in the choice of different strategies for the updating algorithm.

In the model the file HEAD is described by giving as input the number of head blocks M in one head block set and the size of each head block. When simulating the updating process this is divided into two parts, file growth, i.e. the growth of posting sets associated with records already in HEAD; and the insertion of new records.

The expansion factor of the already existing posting sets is drawn randomly from a uniform probability distribution with mean value as a model parameter.

The new records to be added to HEAD are simulated by three stochastic variables H_0, S and NBITS. H_0, corresponding to the address of the home block and S, a parameter of the step algorithm (see appendix B), are drawn randomly from the integers 1,2,...,M. The size of the new head NBITS is chosen from the probability distribution function F(x).

Experiments with the MEDLINE inverted file have shown that F(x) could be represented by a fairly simple linearized function, which is given as input to the model.

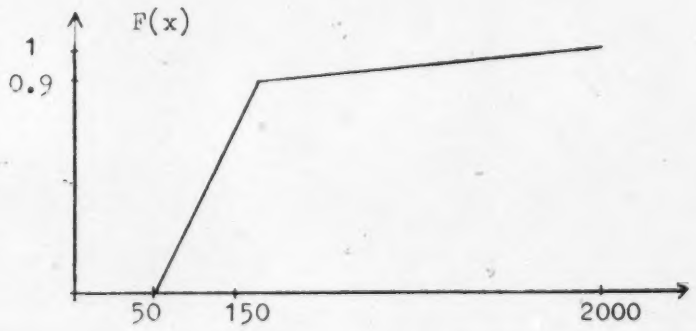


Figure 1 The probability distribution function F(x).

For the particular F(x) illustrated in figure 1 and with M = 64 and a head block size of 18000 bits we get the values:

load factor = 0.97

new records stored in home block / new records = 0.94

The program stops when there is no space available to store the representation of the posting set of a record in any of the 16 head blocks examined. This means that the possibility to remove records from HEAD, thus creating space is not considered. We conclude from the experiment that it is possible to get a high load factor without running through this more time consuming part of the updating algorithm. We also get a high proportion of records that are stored in their home blocks and therefore are retrievable in one access.

For inverted files, with a different probability distribution function, the simultation program can be used to design the file structure and choose appropriate and efficient updating strategies. It should also be possible to predict performance, in terms of load factor and average number of accesses needed to retrieve a record.

REFERENCES

1) S.T. Byrom and W.T. Hardgrave: Representation of Sets on Mass Storage Devices for Information Retrieval Systems {*National Computer Conference*, 1973, pp 245-250}

2) Cheng-Kwei Chou: *Algorithms for Hashcoding and Document Classification* {University of Illinois, 1972}

3) J. Hultgren and R. Larsson: *PASTIME - A System for File-Compression* {Report TRITA-LIB-4043, April 1975}

4) IBM, Storage and Information Retrieval System/VS Program Reference Manual, SH-5400-1

5) R. Larsson et al.: *3RIP-Preliminary Study* {SINFDOK Report 732071 S, 1973}

6) D. Lefkovitz: *File Structures for On-line Systems* {New York, Spartan Books 1969}

7) D. Murray: A Scatter Storage Scheme for Dictionary Lookups {*Scientific Report* No. ISR-16, Department of Computer Science, Cornell University, N.Y. 1969}

8) Software Sciences, ROBOT - an Independent DBMS (*Data Processing* 5:5 (1973)}